Documentation

This was created as a take home task for the position of RSE.

Summary

The program accepts two Roman Numerals from the command line, sums their numerical values, and outputs the sum as a valid Roman Numeral.

Limitations

The input is regulated on the algorithmic level. The input is considered a string of length 20, is not null and does not include non-valid characters.

The permitted Roman Numerals as input and output are ranged from I to MMMCMXCIX (1-3999). However, the output can take values beyond 3999, and while they don't conform to the valid Roman Numeral rules, they conform to the same rules as on the range of 1-3999, except the addition of more than three M. For example, the numerical value of 6999 would read as MMMMMMCMXCIX.

Compilation flags

The source code compilation in Windows is performed by Intel Fortran Compiler x64 v17. The following compilation flags are used in Windows (W) and Linux (L).

- /nologo (W)/ -nologo (L): DEFAULT: enabled. It removes extraneous compiler version output from the command line.
- /warn:all (W)/ -warn all (L): DEFAULT: disabled. Specifies diagnostic messages to be issued by the compiler. It is enabled during development and testing, and shows all compiler errors, critical and non-critical.
- /check:all (W)/ -check all (L): DEFAULT: disabled. It is enabled during development and testing, and allows the compiler to check if loops go out of bounds at run time.
- /traceback (W)/ -traceback (L): DEFAULT: enabled. Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.
- /Qipo (W)/ -ipo (L): DEFAULT: enabled. Enables interprocedural optimization between files. The compiler performs inline function expansion for calls to functions defined in separate files.
- /fast (W)/ -fast (L): DEFAULT: enabled. Forces the compiler to use higher levels of optimisations.

List of files:

- **ascii.dat**: contains a call to RomanNum I <ASCII>, where ASCII represents 256 ASCII characters. Is used by runascii.bat to test a mix of valid and invalid input and the resulting output.
- **ascii.f90**: contains the source code to create ascii.dat.

- **modules.f90**: contains the module structure PARAMETERS used in the main program, where constants are located. The compilation first creates an object file (modules.obj), and then uses it to compile RomanNum.exe.
- **qrun.bat**: calls RomanNum I <Roman Numeral>, where Roman Numeral is every numeral from I (1) to MMMCMXCIX (3999).
- **RomanNum.f90**: Main program source code.
- **rons.txt**: contains all calls used in qrun.bat.
- **run.bat**: compiles modules.f90 and RomanNum.f90 into RomanNum.exe.
- **runascii.bat:** calls ascii.dat.

List of functions and subroutines

1. **BUFFER_TO_STRING (*buffer, stringValue*):**

This subroutine accepts a character input assigned to *buffer*, adjusts all characters to the left, and removes any trailing spaces. Then it assigns the remaining characters to *stringValue*, with a deferred length equal to the actual length of the string.

The subroutine will produce an error if the resulting *stringValue* is null. The program will stop.

2. **CHECK_STRING_VALID (*stringValue*):**

This subroutine accepts a *stringValue* of deferred length and checks whether each character is valid or invalid. Valid characters include all Roman Numeral characters, in both upper and lowercase, as seen below:

$$I, V, X, L, C, D, M$$

$$i, v, x, l, c, d, m$$

Invalid characters are considered all ASCII characters that are not the above, and they will produce an error. The program will stop.

3. **COUNTING (*order, counters*):**

This subroutine accepts the *order* parameter which is an integer that denotes the position of a numeral in ascending order of value.

$$I, IV, V, IX, X, XL, L, XC, C, CD, D, CM, M$$

Also, it accepts the *counters* array which has a number of elements equal to the number of numerals shown above. The array stores the times each numeral has been detected. Then, each counter is compared to the total amount of valid counts for each numeral, stored in the public array *validCounts*. When a counter is larger than its valid amount, the numeral succession given is invalidated and an error is produced. The program will stop.

4. **NUMBER_TO_NUMERAL (*numberArabic*) RESULT (*numeral*):**

This function was sourced from https://rosettacode.org/wiki/Roman_numerals/Encode#Fortran. It accepts an integer number and returns the equivalent Roman Numeral.

The initial number is divided iteratively, by the descending order of the numeral values, performing an integer division. If the result is non-zero, the remainder is ignored, and the result is then used to collate the equivalent numeral equal times. If the result is zero, the next numeral value is used for the division. At the end of the step, the number is update by subtracting the numeral value times the result of the division. The process is followed iteratively, until the number is zero.

5. **NUMERAL_TO_NUMBER (*numeral*) RESULT (*numberArabic*):**

This function was sourced from https://www.rosettacode.org/wiki/Roman_numerals/DECODE#Fortran. It accepts a string of a combination of Roman Numeral characters and either returns an error if the combination is invalid, or the correct numerical value if the combination is valid. The initial form sampled had several problems with its implementation, including insufficient testing for the proper combination of numerals, insufficient testing for double value numerals, and insufficient testing for the total valid count of any individual numeral. All these problems were corrected in this version.

The function accepts the string and starting from the last character it iterates towards the first character. In the start of the step, if the previous value tested is part of a double value numeral that means that the current value is as well. The program then resets the double value numeral flag to FALSE and skips the current step through the CYCLE command. This is explained in depth during the distinction between single and double value numerals.

The proper numeral selection is then enforced by a switch structure, the SELECT CASE (*selector*) structure. The *selector* is specified as a one-character sub-string of the original string. Built in the switch structure are specific cases for each valid value of the *selector*. The valid values are as follows:

$$I, V, X, L, C, D, M$$

When the algorithm detects one of them, it first sets the number value accordingly and then it uses the COUNTING subroutine to increase the count of that numeral by 1. However, the aforementioned selection fails to identify the double value numerals (which was also a problem in the initial sourced implementation). To rectify this problem, each *selector* encloses two additional conditional checks; whether the current numeral position is the last iteratively (so there is no other value beyond), and whether the numeral that iteratively follows is a valid double value numeral. If the answer to both conditional checks is positive, the number value is reset to the proper value accordingly, the COUNTING subroutine increases the count of that double value numeral by 1 and sets the double value numeral flag to TRUE. However, the algorithm has now counted the current numeral twice, so it decrements the first count by 1 to keep consistency. An additional provision has also been added to prevent invalid numeral sequences where the first component of a double value numeral is also present individually. Take for example the numeral CMC. MC does not constitute a double value numeral, so C will amount to 100. However, CM does, and amounts to 900. The sum of the two would be 1000, which should be M and not CMC. Essentially, the provision prohibits two numerals to coexist, one of which is double valued, if their sum would nominally be another numeral. If the conditional returns TRUE, an error is produced, and the program stops.

For visualization purposes, the *selector* choices and their accompanying double value numerals are shown below:

$$I, (\ IV, V), (IX, X), (XL, L), (XC, C), (CD, D), (CM, M)$$

To update the number, first a conditional check is performed on whether the new value is equal or larger than the previous value, which should nominally be the case as the iterations are backwards, from smaller to larger numeral values. If the conditional returns TRUE, the number is increased by adding the new value.

However, there are two conditional provisions in place before the program can continue. The first conditional check intends to capture invalid double value numeral succession. Consider the numeral IXIV. The distinct counters for each I component will allow this form to go past the *selector* while it is invalid. This anomaly is true for all double value numerals with a real ratio of 4/9, as shown below:

$$\frac{IV}{IX} = \frac{XL}{XC} = \frac{CD}{CM} = \frac{4}{9}$$

The conditional check returns true if such a ratio is true, it returns an error and stops the program. The second conditional check intends to capture the mixing of double value and single value numerals of the same order. Consider the numeral IXV. Its form while invalid, has not tripped any other conditional checks. The conditional check returns true if a double value numeral is present and its real value is less than twice the previous numeral’s real value. An error will be produced, and the program will stop.

At the end of the step, the previous value is updated with the new value, in preparation of the new step.